AIAA-98-4160

SIMULATION OF A F/A-18 E/F DROP MODEL USING THE LaSRS^{++} FRAMEWORK

Kevin Cunningham[*], P. Sean Kenney, Richard A. Leslie
David W. Geyer, Michael M. Madden[*], Patricia C. Glaab

Unisys Corporation
NASA Langley Research Center
MS 169
Hampton, VA 23681

Abstract

A simulation of a 22% dynamically scaled F/A-18 E/F Drop Model was successfully developed within the Langley Standard Real-Time Simulation in C^{++} (LaSRS^{++}) framework. Development in the LaSRS^{++} framework is done using object-oriented analysis, design and programming techniques. Common software design patterns are also used. Development using the LaSRS^{++} framework promotes the development of a simulation which is inherently maintainable, extensible, reliable and computationally efficient.

Introduction

The goals of maintainability, extensibility and reliability are certainly common to all software development efforts. The use of object-oriented analysis, design and programming techniques to meet these goals is well established.[1–4] What is not well established is the application of these techniques to the development of a piloted flight simulation that must necessarily operate in a hard real-time environment. These techniques were used to develop a maintainable, extensible and reliable F/A-18 E/F Drop Model Simulation for the Simulation Systems Branch at NASA Langley Research Center. Furthermore, the unusually strict demands for computational efficiency imposed by the simulation of dynamically scaled models were met with ease.

The F/A-18 E/F Drop Model Simulation was developed in support of the Drop Model Program at NASA Langley Research Center. The 22% dynamically scaled model is flight tested to support aerodynamic and flight control research interests. As part of the test technique,[5] the unpowered model is carried to its release altitude by helicopter. After release from the helicopter, a pilot on the ground uses down-linked drop model cockpit video and sensor data to fly the flight profile. Commands from the crew station, along with the down-linked data are used by the ground based flight control computer to calculate control surface actuator commands. The resulting commands are up-linked to the model. At the end of the flight, a parachute is deployed and the model recovered.

For the drop model to achieve an accurate representation of the full scale aircraft's flight dynamic characteristics, dynamic scaling techniques must be applied to the model.[6] The geometric and mass properties of the model are dynamically scaled by matching the Froude number. The same scaling technique must be applied to the temporally dependent aspects of the flight control laws as well as the sensed data used in the control laws. The scaling technique dictates that the model's time scale is inversely proportional to the square root of

[*] Senior Member, AIAA

the geometric scale factor. Thus, for a 22% dynamically scaled model, the control laws must run at a frame rate over 200% faster than that of the full-scale aircraft. For this reason, the implementation of dynamically scaled control laws imposes unusually high demands for computational efficiency.

This F/A-18 E/F Drop Model Simulation is primarily used for evaluation and tuning of control laws, flight profile planning and crew training. Additionally, the implementation of the flight control laws used by the simulation serve as a verification model for the Drop Model Program's independent implementation of the same laws. The simulation, developed using the GNU C^{++} compiler, supports pilot-in-the-loop, synchronous real-time operations with integration rates in excess of 300 Hz on SGI Onyx systems at NASA Langley Research Center.

Fundamental Concepts

The fundamental concepts behind the LaSRS^{++} framework[7] are:

- simplification of behavior (via abstraction)
- protection of data (via encapsulation).

The use of object-oriented C^{++} and a few simple implementation guidelines within the framework create an environment where these concepts are not only simply allowed, but fostered. Simplification of behavior is primarily achieved by abstraction of detail. When details are handled elsewhere, the user need not be distracted by the complexities at hand. In object-oriented design, this takes on a layered form. Once the desired levels of abstraction are decided upon, these various levels of complexity are represented by separate classes.

Classes are the fundamental building blocks in C^{++}. A class usually represents a single concept or physical entity. A developer specifies the behaviors (functions) and the data that a class will contain. An object is a particular instance of a class. An object named "Titanic" would likely be a particular instance of a doomed ocean-liner class. In this regard one can think of a class as they would think of an intrinsic data type (int, float, double). One object that creates a second object is said to contain that object. This is often referred to as a "has a" relationship. (e.g. An F/A-18E has a flight control system.)

Inheritance is the mechanism by which the details from one layer are added to another. The inheritances may be chained together. Inheritance is often referred to as an "is a" relationship (an F/A-18 is an airplane). Thus, a class representing an F/A-18 would not contain all the aspects that are common to all aircraft, just the aspects that make an F/A-18 unique. Inheritance is the primary means for the abstraction of complexity.

Polymorphism is one of the most powerful features of object-oriented languages. It is the means by which a behavior declared in a parent class may be defined differently by each child class. It allows different behavior to be realized via a common interface. This simplifies software and promotes the use of common interfaces.

The C^{++} language provides a means to allow or disallow a client's access to the behaviors and data in a class. Public access (any client has access) and private access (no client has access) provide the two extremes of data/function protection. A significant feature of the LaSRS^{++} framework is that all data, in all classes of the framework is private. Containment ("has a") or inheritance ("is a") relationships are not permitted to break this data encapsulation. If the developer of a class wishes to allow outside access to the data in a class, accessor functions with the appropriate level of protection (public or protected) must be written to reference the data. The data access provided may be limited to read-only or, if required, both read and write. In this manner, the LaSRS^{++} framework provides excellent data protection from accidental corruption through the use of strong encapsulation.

The Framework

The F/A-18 E/F Drop Model Simulation was developed within the LaSRS^{++} framework. A framework is a collection of classes which are used to build various products. This framework is kept under strict configuration control. The classes contained therein are carefully reviewed with regard to both design and implementation.

Classes in the LaSRS^{++} framework[7] can be divided into two major categories. The first is very general in nature and serves as a "toolbox" which contains classes for:

- filtering
- function table lookups
- vector and matrix operations
- statistical and random variable calculations
- memory management[8]
- input - output capabilities[9]
- data conversions

The second is oriented more toward simulation and contains classes relating to:

- equations of motion
- propulsion systems
- flight control systems
- weapons systems
- aerodynamic modeling
- environmental modeling
- mass property modeling
- multi-vehicle, multi-cpu simulation[10]
- hardware interfacing[11]
- navigation systems
- trim capabilities

Using The Framework

Two classes of interest in the framework are named Vehicle and Aircraft. These classes are shown, along with their relationship to an F/A-18 E/F Drop Model class in Figure 1.

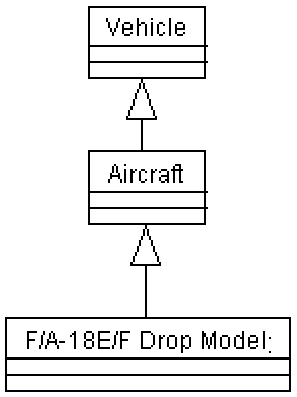

Figure 1: Inheritance in action

The diagram presented in Figure 1 is a class diagram using the Unified Modeling Language.[12] It makes the simple statement that the F/A-18 E/F Drop Model class inherits from Aircraft which inherits from Vehicle. (An F/A-18 E/F Drop Model "is a" Aircraft, which "is a" Vehicle.) The Vehicle class contains data such as acceleration, velocity and position vectors. It also contains functions for the calculation of these quantities. Aircraft contains data such as angle of attack, side slip, calibrated airspeed and functions to calculate these quantities.

The F/A-18 E/F Drop Model class need only take responsibility for registering its unique attributes with the Aircraft and Vehicle classes. Since the data in those classes only allows private access, this is implemented using in-lined mutator functions. These mutator functions would typically have only protected access (only a sub-class may use the function). The use of the C^{++} `inline` keyword plus compiler optimizations make the use of protected mutator functions a very efficient means of controlling access to data. In addition to defining its attributes, the F/A-18 E/F Drop Model class must also define its unique behaviors. In practice, this primarily consists of a few behaviors that update the forces and moments. The forces and moments are then registered with the vehicle, and the inherited behavior to perform integrations is invoked. All of the complexity associated with integrating the state vector is hidden from the developer.

A high-level representation of several of the major components of the F/A-18 E/F Drop Model is shown in Figure 2. This class diagram shows the use of aggregation and inheritance from the framework. The aggregation would be verbalized as “the F/A-18 E/F Drop Model has an F/A-18 E/F Drop Model Sensor System”, etc. The solid black diamonds in Figure 2 indicate that the F/A-18 E/F Drop Model class not only contains these classes but is responsible for their creation and destruction. The use of aggregation provides another means to logically remove complexity and encapsulate data while modeling a system. In this example, the inheritance from the framework establishes a common interface to be used by the sub-classes. It is then the responsibility of an F/A-18 E/F Drop Model Simulation developer to provide these details, as they could not be known at the framework level.

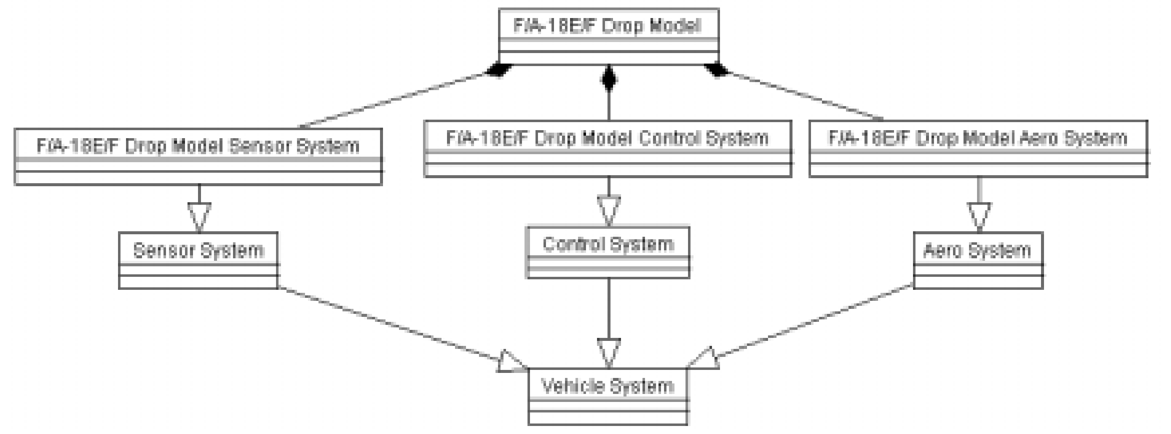
F/A-18E/F Drop Model
F/A-18E/F Drop Model Sensor System
F/A-18E/F Drop Model Control System
F/A-18E/F Drop Model Aero System
Sensor System
Control System
Aero System
Vehicle System

Figure 2: High-Level View

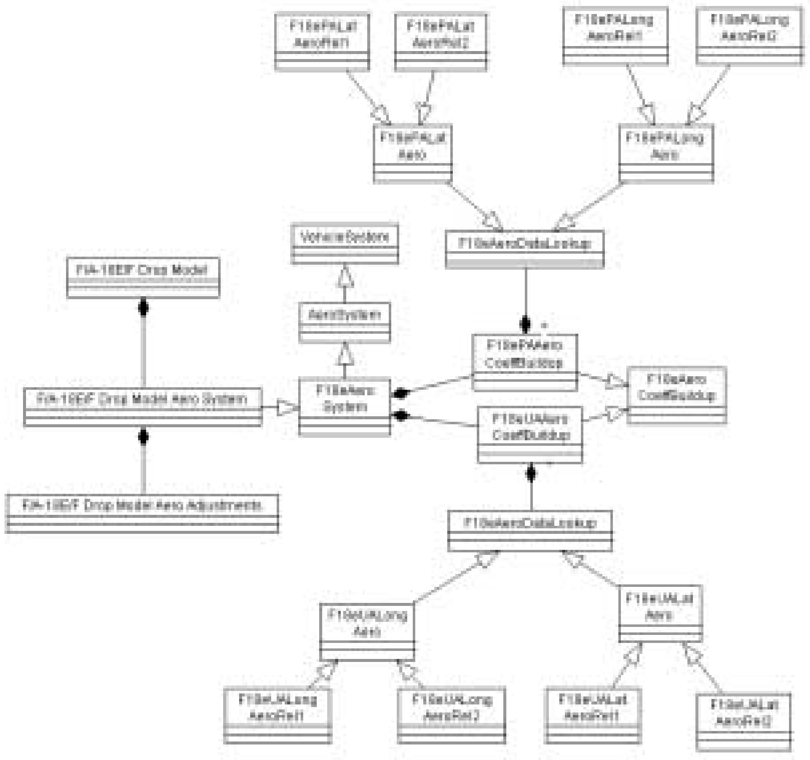

Figure 3: Essentials of the Aerodynamic Model Design

An Object-Oriented Aerodynamic Model

The simulation's aerodynamic model uses two separate databases. One is used for an "up and away" flight regime and the other for a "powered approach" flight regime. Thus, it is necessary to switch between the two databases during synchronous real-time. A class diagram with the details of the aerodynamic model is shown in Figure 3.

Figure 3 shows that the F/A-18 E/F Drop Model has an F/A-18 E/F Drop Model Aero System, which is an F/A-18 E/F Aero System. The only aspects not inherited directly from the full-scale aircraft's aerodynamic model is F/A-18 E/F Drop Model Aero Adjustments. These adjustments represent the differences between the aerodynamic model and the performance of the model being tested. The F/A-18 E/F Aero System has two aero data lookup objects ("F18ePAAeroCoeffBuildup" and "F18eUAAeroCoeffBuildup"). The F/A-18 E/F Aero System acts as a "mediator" between the vehicle and these aerodynamic coefficient build-up objects. The F/A-18 E/F Aero System gets data from the vehicle and passes it on to the appropriate coefficient build-up object, instructs that object to perform its coefficient build-ups and obtains the resulting force and moment coefficients.

These two aerodynamic coefficient build-up classes contain the details of the aerodynamic model. The summation of the aerodynamic coefficients which contribute to final force and moment coefficients is defined therein. The calculation of these coefficients is performed within the aerodynamic data lookup objects that each of the aero coefficient build-up objects contain. The details involved in performing the aerodynamic lookups are grouped into logical sub-classes (e.g. longitudinal and lateral-directional coefficients). The only complexity remaining is the actual data. These classes, diagramed in Figure 3, are represented with a release number (Rel1, Rel2). Subsequent updates to the aerodynamic data result in the creation of a new class with the appropriate release number.

The primary goals of this architecture were to increase maintainability, extensibility and overall reliability. It takes no stretch of the imagination to realize that the aerodynamic model of an aircraft undergoing flight testing will be subject to frequent updates. New releases of the aerodynamic model quite often are limited in scope. At best, only the data in one of the leaf classes (the most specific classes in an inheritance hierarchy) would be changed. The only changes that need to be made to the software are limited to the creation of a class representing the new release. There are no monolithic data files to contend with. The source of any errors that do occur must reside in the new class. The use of aggregation in the design allows the model to be easily extended. Aerodynamic models for any new aspects of the aircraft could be easily handled as a new aggregate of a aerodynamic coefficient build-up class. The interfaces would remain the same and there would simply be another object to service. Through the use of inheritance and aggregation, complexity is dealt with in layers and as smaller, more manageable components which greatly simplifies any aspect of the model.

The Mediator Design Pattern

There is much commonality in the way certain objects interact. These patterns are repeated over and over regardless of the system being modeled. Establishing efficient and effective mechanisms for these relationships is a problem which requires careful attention. Design patterns are evolved, yet simple solutions to these common problems in software development.[2] By using “tried and true” design patterns, a software developer is spared the time and expense of iterating to an efficient solution to the problem.

Figure 4 illustrates the interdependencies that could result when components of a flight control system share data through direct interaction. Each class is then dependent on all the other classes that it needs data from. This creates a tightly coupled system. Any new data or behavior added to one class affects all the other classes that depend on it. As a tightly coupled system grows, it tends to take on the characteristics of a monolithic class. This limits re-usability, hampers testability and significantly increases the time to compile the software. The solution to this common design problem is the Mediator Design Pattern.

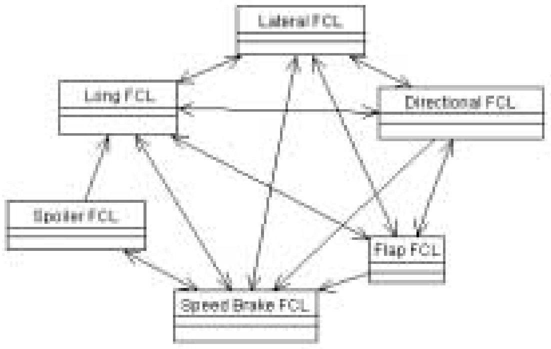

Figure 4: Class Dependencies Abound

The use of the Mediator Design Pattern rids objects of their explicit dependencies. This greatly decouples and simplifies a system. Figure 5 illustrates the impact of a mediator on the system shown in Figure 4.

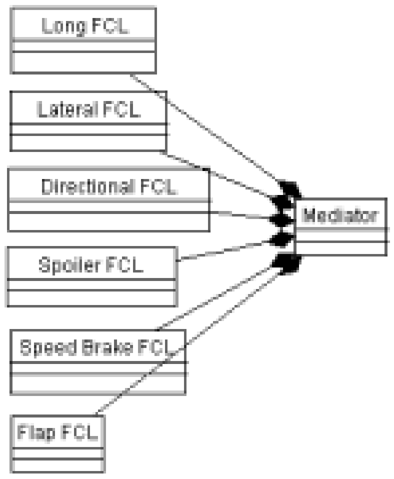

Figure 5: The Mediator Pattern

Figure 5 is the more simple design. It is the superior design. The aggregate classes no longer depend on each other. In fact, they do not even depend on the mediator class which encapsulates them. This autonomy greatly increases re-usability. Testability is increased in that each class may be tested as a single unit, rather than testing the whole system at once. Finally, compilation times are reduced by the fact that a software change that only affects one class will only require re-compilation of that class, not an entire coupled system.

The use of the Mediator Design Pattern in the F/A-18 E/F Drop Model Simulation is shown in Figures 2 and 3. The F/A-18 E/F Drop Model system classes for the aerodynamics, controls and sensors all serve in the role of mediator. These mediator classes take the responsibility for passing data between and managing the behavior of their component objects.

Flight Control Laws and Polymorphism

Figure 6 shows more details of the F/A-18 E/F Drop Model Simulation flight control laws. The features to note are the inheritance relationships from the full-scale F/A-18 E/F classes. The behavior of the F/A-18 E/F Drop Model control laws are for the most part directly inherited from the full-scale implementation. However, there are some aspects of the full-scale control laws that are not used by the drop model.

An example of this can be found in the directional axis control laws. The full-scale laws compute a nose wheel steering command when the control laws are instructed to update. However, such a computation is unnecessary for a drop model, as it has no landing gear. The problem is then to be able to inherit the overall behavior of the control laws, while redefining these kinds of behavioral differences. In a procedural paradigm this would be handled in the baseline code with logical branching. However, as time goes on and other projects use the baseline functionality, this logic would likely evolve into a maintenance nightmare. Within the LaSRS^{++} framework this sort of behavioral change easily handled thru polymorphism.

The base class (with the baseline set of behaviors) need only define a virtual (the C^{++} keyword which invokes polymorphic behavior[13, 14]) function representing the behavior. In the sub-class, the same virtual function need only be redefined with the desired behavior. The sub-class can still inherit the fundamental behaviors of the control laws. Only the behavioral aspects that have been polymorphically redefined will change. Once a polymorphic behavior is defined in the base class, it may be easily extended an infinite number of times. Using polymorphism to extend behavior in this manner provides infinite extensibility without creating maintenance burdens. Projects other than the F/A-18 E/F Drop Model could just as easily inherit from the base class and redefine certain behaviors. All sub-classes would be isolated from each other, while at the same time, receiving changes made to the base class.

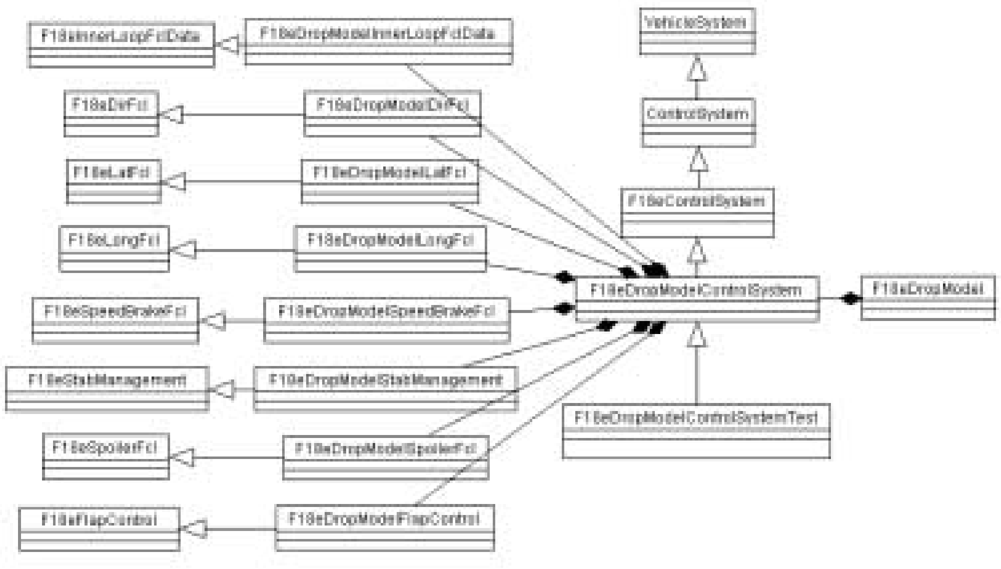

Figure 6: Drop Model Control Law Architecture

Conclusions

A simulation of a 22% dynamically scaled F/A-18 E/F Drop Model was successfully developed using, not only the software, but also the architectural principles of the Langley Standard Real-Time Simulation in C^{++} (LaSRS^{++}) framework. This simulation's architecture makes frequent use of inheritance, aggregation, polymorphism and the mediator design pattern. This architecture provides strong encapsulation of data while promoting maintainability, extensibility, reliability and re-usability. The high integration rates achieved by the F/A-18 E/F Drop Model Simulation demonstrate the computational efficiency of the LaSRS^{++} framework.

Acknowledgments

The authors would like to thank Dawn Idema, Stacie Clark and Gus Taylor (Unisys Corporation) for their significant design and implementation contributions to the development of the F/A-18 E/F Drop Model Simulation. Also, Mark A. Croom (Vehicle Dynamics Branch, NASA Langley Research Center) for support, guidance and many technical discussions relating to drop model testing and flight dynamics.

Bibliography

[1] Grady Booch. *Object-Oriented Analysis and Design*. Benjamin/Cummings, Redwood City, California, 1994.

[2] Gamma E., Helm R., Johnson R., Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[3] John Lakos. *Large-Scale C^{++} Software Design*. Addison-Wesley, Reading, Massachusetts, 1996.

[4] Robert C. Martin. *Designing Object-Oriented C^{++} Applications Using The Booch Method*. Prentice-Hall, Englewood Cliffs, New Jersey, 1995.

[5] Mark A. Croom, et al. Dynamic Model Testing of the X-31 Configuration for High-Angle-of-Attack Flight Dynamics Research. Paper Number AIAA-93-3674 CP, August, 1993.

[6] C. H. Wolowicz, J. S. Bowman, W. P. Gilbert. Similitude Requirements and Scaling Relationships as Applied to Model Testing. Technical Report NASA TP 1438, 1979.

[7] Richard A. Leslie, et al. LaSRS^{++} An Object-Oriented Framework for Real-Time Simulation of Aircraft. Paper Number AIAA-98-4529, August, 1998.

[8] David Geyer, et al. Managing Shared Memory Spaces in an Object-Oriented Real-time Simulation. Paper Number AIAA-98-4532, August, 1998.

[9] Patricia Glaab, et al. A Method to Interface Auto-Generated Code into an Object-Oriented Simulation. Paper Number AIAA-98-4531, August, 1998.

[10] Michael Madden, et al. Constructing a Multiple-Vehicle, Multiple-CPU Using Object-Oriented C^{++}. Paper Number AIAA-98-4530, August, 1998.

[11] P. Sean Kenney, et al. Using Abstraction to Isolate Hardware in an Object-Oriented Simulation. Paper Number AIAA-98-4533, August, 1998.

[12] Terry Quatrani. *Visual Modeling With Rational Rose and UML*. Addison Wesley, Reading, Massachusetts, 1998.

[13] Bruce Eckel. *Thinking in C^{++}*. Prentice-Hall, Englewood Cliffs, New Jersey, 1995.

[14] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, Massachusetts, third edition, 1997.

[15] Scott Meyers. *Effective C^{++}*. Addison-Wesley, Reading, Massachusetts, second edition, 1998.

[16] Scott Meyers. *More Effective C^{++}*. Addison-Wesley, Reading, Massachusetts, 1996.

[17] Atul Saini David R. Musser. *STL Tutorial and Reference Guid*. Addison-Wesley, Reading, Massachusetts, 1996.